2.3.2021

BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPT.

CS202 HOMEWORK 2 SECTION 2

BINARY SEARCH TREES

ZÜBEYİR BODUR

21702382

Spring 2021

1. Question 1

a) Part a

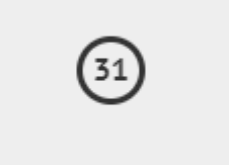
Prefix : / * A + B C D

Infix : (A * (B + C)) / D

Postfix : A B C + * D /

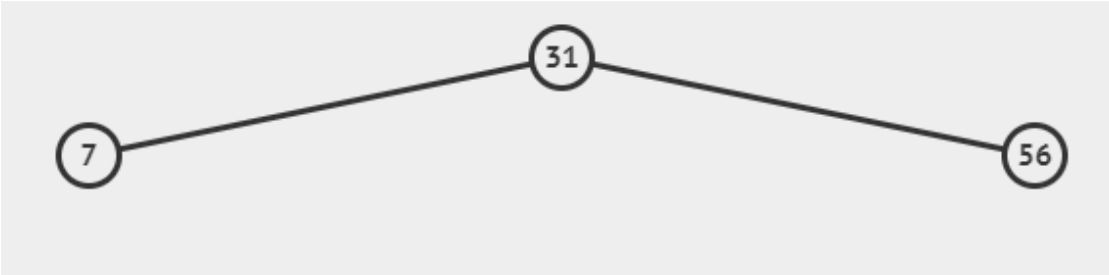
b) Part b

Insert 31 :

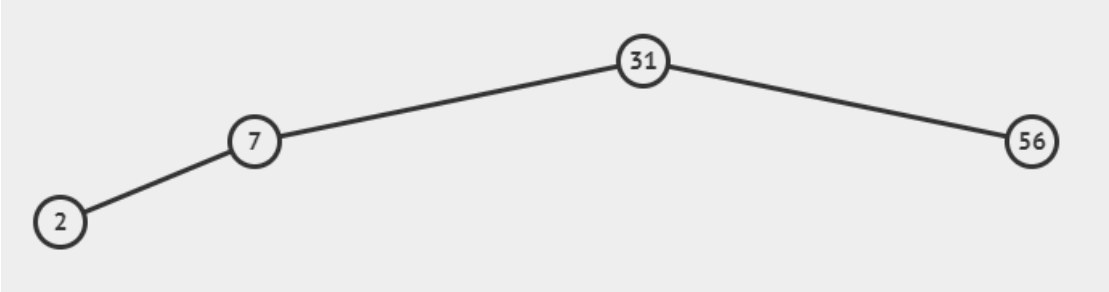


Insert 7 :

Insert 56 :



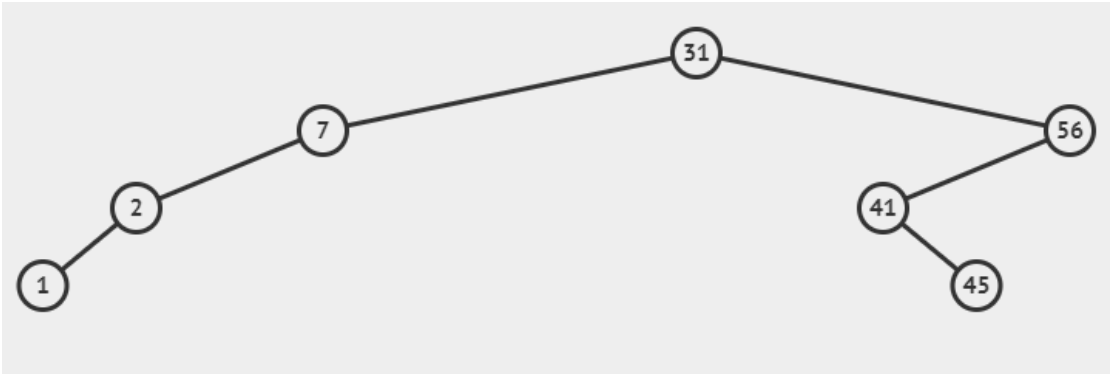
Insert 2 :



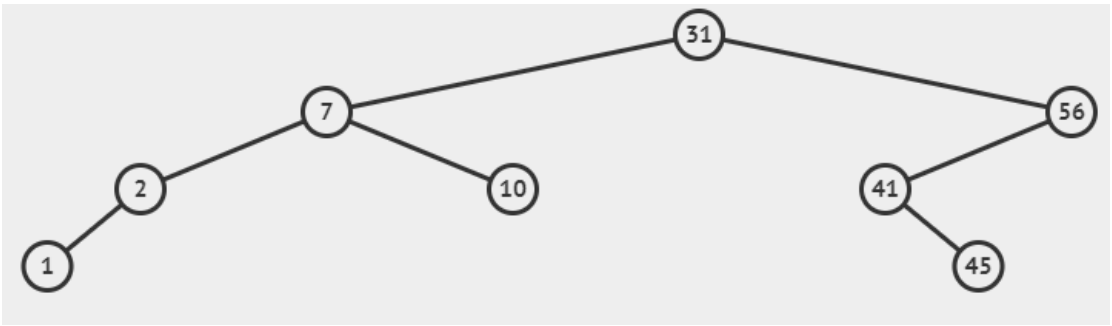
Insert 1 :

Insert 41 :

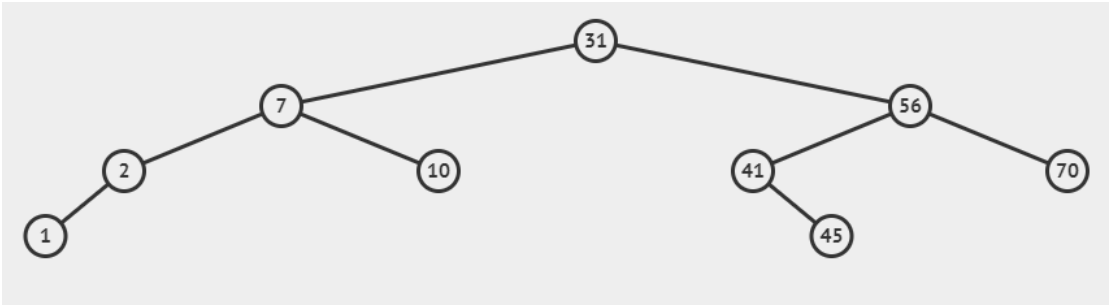
Insert 45 :



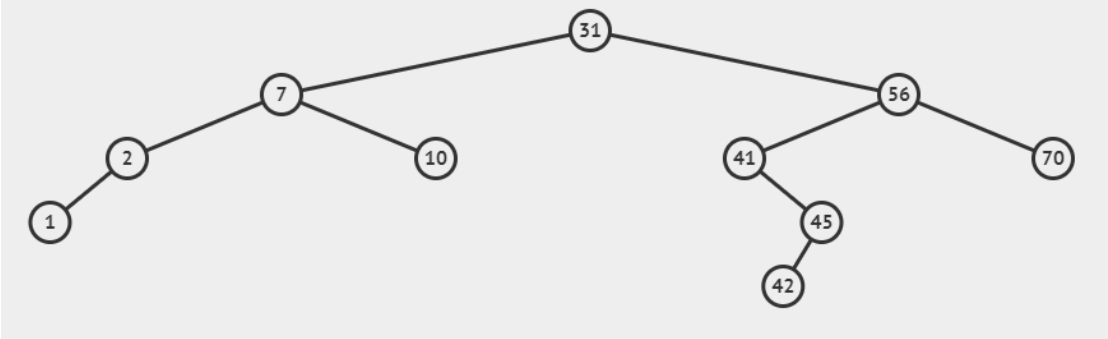
Insert 10 :



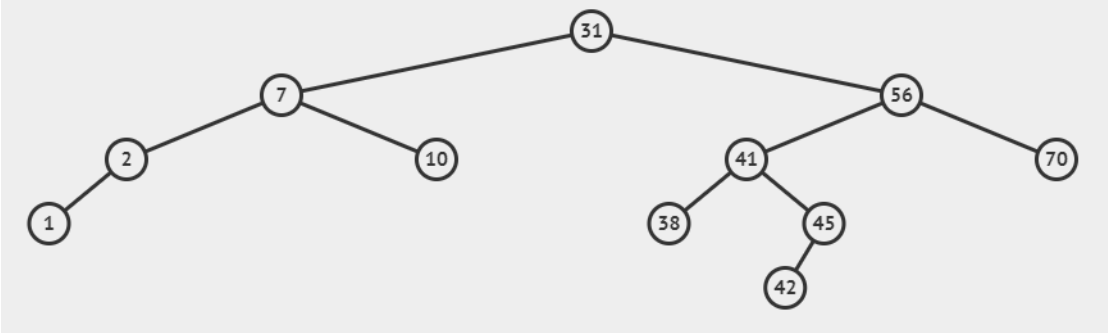
Insert 70 :



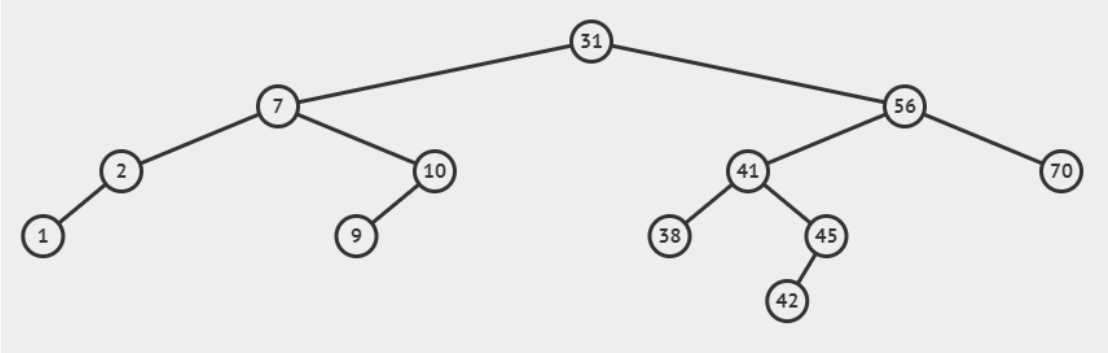
Insert 42 :



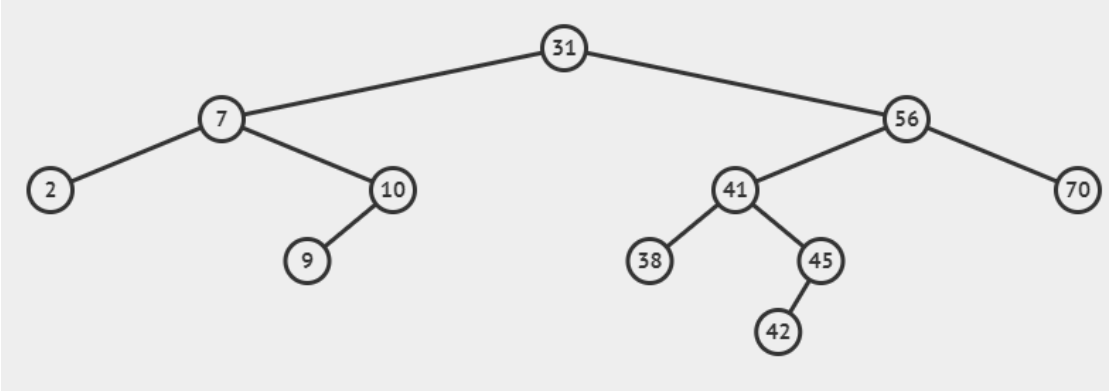
Insert 38 :



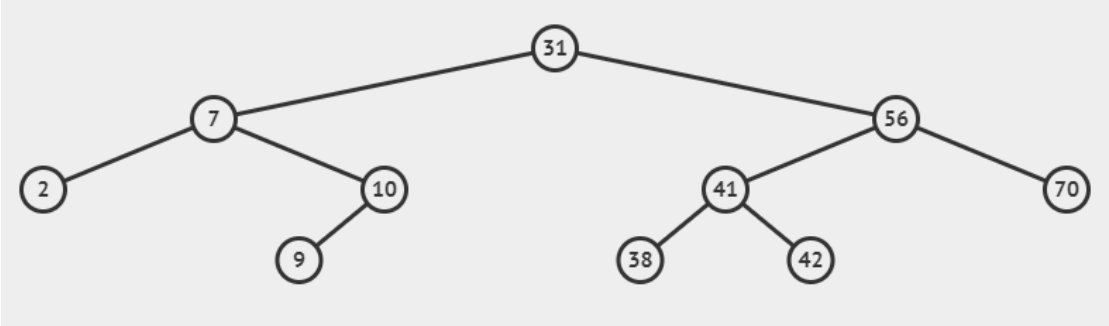
Insert 9 :



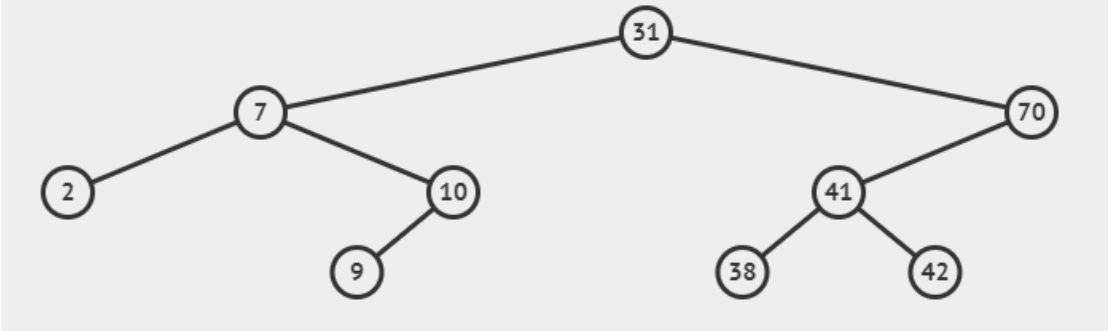
Delete 1 :



Delete 45 :



Delete 56 :



Delete 7 :

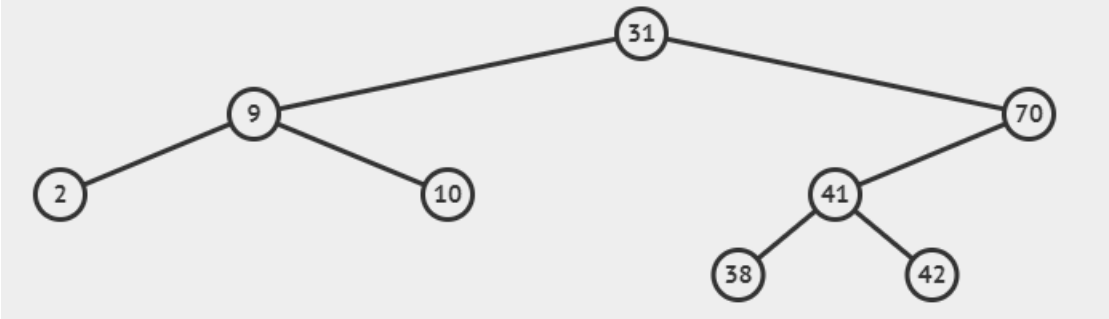


Figure 1-16 : Evolution of the BST for each insertion & deletion operation

c) Part c

The preorder traversal of the tree is :

18, 5, 11, 9, 7, 13, 12, 15, 21, 19, 28, 23, 25, 24, 26

Then, the tree will have the root 18, and the corresponding tree will be the following:

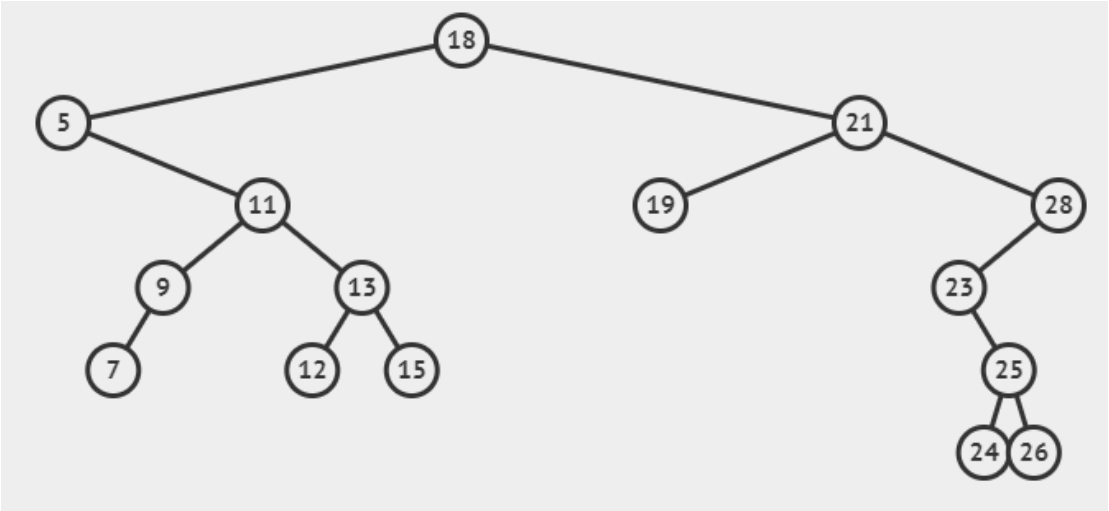


Figure 17 : Corresponding BST for the given preorder traversal

Postorder traversal of the tree will be the following:

7, 9, 12, 15, 13, 11, 5, 19, 24, 26, 25, 23, 28, 21, 18

2. Question 3

```
void BinarySearchTree::levelorderTraverse(btn *&node) {
    NodeQ queue;
    // set the root as the starting point
    if (node)
        queue.enqueue( & node);
    while (!queue.isEmpty()) {
        // mark tmp as visited
        BinaryTreeNode* tmp = queue.dequeue();
        cout << tmp->data << " ";
        // visit tmp's children
        if (tmp->left)
            queue.enqueue( & tmp->left);
        if (tmp->right)
            queue.enqueue( & tmp->right);
    }
}
```

Figure 18 : Implementation for levelorderTraversal function

The leveorderTraverse function uses breadth-first traversal (BFT) method to visit all nodes, unlike inorder/preorder/postorder traversals. It visits each node only once. Even though it's not visible from the source code, since we know that each node is visited only once, we can just say that levelorderTraverse is O(n).

Implementation of such traversal can't be made asymptotically faster as it's a traversal at the end, meaning each node needs to be visited at least once. However, we can implement it slower if we were to use a different algorithm other than BFT.

The function can't be implemented asymptotically faster as all the nodes that is enqueued will be dequeued. Choosing front or back as head in linked list implementation of queues doesn't make the algorithm faster.

```
int BinarySearchTree::span(btn *&node, const int a, const int b) {
    if (node) {
        int left = 0, right = 0;
        if (node->data < b)
            right = span( && node->right, a, b);
        if (node->data > a)
            left = span( && node->left, a, b);
        return left + right + ( (node->data >= a && node->data <= b) ? 1 : 0);
    }
    return 0;
}
```

Figure 19 : Implementation for span function

Span function, as asked, doesn't visit every node in a given binary search tree. It visits a node if it's parent is in span of [a, b] closed interval. If a node is not in this interval, number of nodes for the span [a, b] in this sub tree is immediately set to 0. Because we don't need to look the remaining elements of such a sub tree, as it is a binary search tree, it will be already outside the interval as a whole.

The time complexity of this function is O(n) in the worst case, that is whole tree is inside the interval [a, b]. If this is the case, we would need to visit ever node in the tree, giving us O(n) worst case time complexity.

The function can't be written asymptotically faster as we already don't traverse the unnecessary parts of the tree.

```
void BinarySearchTree::mirror(btn *&node) {
    if (node) {
        btn* tmp = node->left;
        node->left = node->right;
        node->right = tmp;
        mirror( &: node->left);
        mirror( &: node->right);
    }
}
```

Figure 20 : Implementation for mirror function

Mirror function swaps the left sub tree and right sub tree of a given binary search tree in preorder. After a swap operation is performed, sub trees also need to be mirrored to complete the operation.

As it uses the exact same approach as preorder traversal, it visits every node only once. Hence, the worst case time complexity for the swap function will be O(n).

The function can't be written asymptotically faster as we must visit every node to mirror a given tree. Moreover, a swap operation requires three moves, we can't implement a swap operation that has less than three moves, say two moves. However, if we were able to do so, the function would be approximately 1.5 times faster.